Perception in Robotics
PS3

Gonzalo Ferrer
Skoltech

27 February 2024

This problem set is a single task comprising 15% of your course grade, and it is to be done individually. You are encouraged to talk at the conceptual level with other students, discuss the equations and even the results, but you may not show/share/copy any non-trivial code.

Submission Instructions

Your assignment must be received by 11:59 p.m. on Sunday, March 10th. You are to upload your assignment directly to the Canvas website as two attachments:

1. A PDF with the written portion of your document, solving the tasks proposed below. Scanned versions of hand-written documents, converted to PDFs, are perfectly acceptable. No other formats (e.g., .doc) are acceptable. Your PDF file should adhere to the following naming convention: `alincoln_ps3.pdf`.
2. A `.tgz` or `.zip` file *containing a directory* named after your uniqname with the structure shown below.

```
alincoln_ps3.tgz:
alincoln_ps3/run.py
alincoln_ps3/field_map.py
...
alincoln_ps3/slam/* //new files created by you
alincoln_ps3/sam.{avi,mp4}
```

Homework received after 11:59 p.m. is considered late and will be penalized as per the course policy. The ultimate timestamp authority is the one assigned to your upload by Canvas.

SLAM problem set

Code

To begin, you will need to pull the latest github version to get the `PS3` folder. This folder contains some Python files. The project is a modified version of the localization simulator originally used in PS2. The number of landmarks is now configurable and multiple range/bearing/markerId tuples are produced at each time step. The odometry model remains the same as in PS2.

Below you will find descriptions of the files included in the zip file. You may end up not using every single function. Some are utilities for other files, and you don't really need to bother with them. Some have useful utilities, so you won't have to reinvent the wheel. Some have a full description in their corresponding files.

Things to implement

- Include all those necessary calls on the `run.py` file to update and correct the SLAM problem. Plotting is also necessary.
- Implement graph SLAM with known correspondences using the mrob library. Remember to `pip install mrob`.

Utilities (Essentially similar from PS2)

- `README.md` – Some commands examples for installing the environment, testing and evaluating the task.
- `run.py` – Main routine, with multiple options, allowing you to solve the task with no need to modify the file.
- `field_map.py` – for plotting the map.
- `tools/task.py` – General utilities available to the optimizer and internal functions.
- `tools/jacobian.py` – Jacobians derived for 2d planar robot and landmark observations.
- `tools/data.py` – Routines for generating, loading and saving data.
- `tools/helpers.py` – For parsing console line arguments.
- `tools/objects.py` – Data structures for the project.
- `tools/plot.py` – All utilities for plotting data.
- `slam/slamBase.py` – An abstract base class to implement the various SLAM algorithms.

Task 1: Prerequisites to build SAM with known DA (60 points)

In this task, you will implement the prerequisites for landmark-SAM. Your robot is driving around an environment obtaining observations to a number of landmarks. The position of these landmarks is not initially known, nor is the number of landmarks. For now, we'll simplify the problem: when the robot observes a landmark, you know which landmark it has observed (i.e., you have perfect data association).

For this problem, your landmark observations are $[range, bearing, markerId]$ tuples. Your robot state is $[x, y, \theta]$ (cm, cm, radians) and the motion control is $[\delta_{\text{rot1}}, \delta_{\text{trans}}, \delta_{\text{rot2}}]$ (radians, cm, radians).

Your task is to use the `mrob` library to handle the graphSLAM problem and provide a solution.

A. (10 pts) **Constructor.** The base class `sam.py` creates a factor graph by using `graph = mrob.FGraph()` at line 22. In order to correctly initialize the problem, we ask you to add the first node, variable x_0, to the graph. For this, you will need to use the method `graph.add_node_pose_2d(x_0)`. Take a look at help for input required dimensions. This function, and all functions creating node variables, return their corresponding node id key, and requires as input an initial guess of the value, in the form of a np.array of 3 elements.

   Now, we need to set an anchor factor to initialize x_0. For that, use `graph.add_factor_1pose_2d(x0, nodeId, W)`. The inputs are: observation, node id, returned by the previous function `add_node_pose_2d`, and information matrix of the observation. All these values are given in the initial conditions.

   Check that you have actually added the node by using the command `graph.print(True)`, it will print all the information in the graph. If set to false, it will only plot the number of nodes and factors. Include in your report the output from print. Since no optimization has been done, you will observe that the residuals are not initialized (garbage values) and Jacobians are set to zero. State variables should be correct.

   Hint: For these initial tasks, you may want to run a single iteration of the optimizer `python3 run.py -s -f sam -n 1`.

B. (10 pts) **Odometry.** Update the function `sam.predict` (in the file `slam/sam.py`) to add an odometry factor. Also, modify the `run.py` consequently. For this, you need to add a new 2d pose node (see A), without need to be initialized (np.zeros(3)). Later, we will add the new factor corresponding to the odometry observation. We will use the function `graph.add_factor_2poses_2d_odom(u, nodeOriginId, nodeTargetId, W_u)`. Take a look at help. There is an interesting feature about odometry factor, it assumes that the node needs to be initialized given the last node state and the action u, so it does all required calculations without further intervention.

   You need to calculate the covariance in state space and invert to obtain the information matrix `W_u`.

   To check this, use the function `graph.get_estimated_state()`. It returns the list of all nodes estimated. Check out the values before and after adding the odometry factor. Include the information in your report.

C. (10 pts) **Landmark observations.** Update the function `sam.update` to add a landmark factor. Bear in mind that there are several observations per time step. In case that the landmark has not been previously observed, you should add a new landmark node to the graph: `graph.add_node_landmark_2d(np.zeros(2))`. Initialization is again automatic if indicated so in the factor. You may add the factor corresponding to the landmark observation: `graph.add_factor_1pose_1landmark_2d(z, nodeOriginId, nodeLandmakrId, W_z, initializeLandmark=True)`. Take a look at help. If `initializeLandmark = True`, then automatically the value of the landmark node is initialized according to the inverse of the observation function, as explained in L08 SLAM. If false (by default), it simply adds the observation without modifying the value of the landmark node (necessary for all other observation except a new landmark.

   You need to calculate the information matrix from the beta parameters.

   Print all nodes in the graph with poses and landmark.

D. (10) **Solve.** Modify the function `sam.solve` to include the solving routine `graph.solve(mrob.GN)`. This function corresponds to a single iteration of the Gauss-Newton optimization.

   Sketch the resultant graphical model of this problem and print the full graph after optimization. Comment on the results.

E. (10) **Manual solution**. Check the solution by calculating the normal equation outside the mrob library. To do this, some extra care needs to be taken to ensure that the correct matrices are being processed. Rerun the code, this time saving the current vector of states x^0 before optimizing and then solve as explained in part D.

   Hint: The function `graph.get_estimated_state()` returns a list of arrays, each of different size for poses and landmarks, so you want to flatten this into the same vector.

   Now, this solution is calculating a single iteration and evaluating the adjacency matrix and residuals at the linearization point x_0. We will make use of the data structures calculated in the FGraph to run the same single step optimization, so you need to extract the adjacency matrix A by using `graph.get_adjacency_matrix()`.

   How the library handles the adjacency is a little different than explained in class, so instead of pre-multiplying by the information, it generates a block diagonal matrix of information W, which can be obtained with the function `graph.get_W_matrix()`, such that $\Lambda = A^\top WA$. Check that the calculated information matrix coincides with the `graph.get_information_matrix()` from the Fgraph by using numpy matrix norm.

   Hint: For this task (and the following), you may want to run two iterations of the system: `python3 run.py -s -f sam -n 2`.

   Clarification: Since the first pose is added with high certainty (information 10^{12}) an error of several thousand units is expected.

F. (10) Solve the normal equation and compare your obtained solution with the one from `solve()`. The vector in the normal equation is $b = A^\top W r$, where r is the vector of residuals. This is already proccessed by the library and you can obtain directly by `graph.get_vector_b()`. You can use again the numpy norm of the difference between both vector solution, one saved before optimizing and the other after solving the normal equation

$$dx = \Lambda^{-1} b = (A^\top W A)^{-1} A^\top W r$$

Task 2: SAM evaluation (40 points)

For the following task, you will be evaluating the data provided in `slam-evaluation-input.npy`.

A. (5 pts) **Incremental Solution.** At each time iteration, solve the SAM problem. Monitor the current error in the graph at each iteration by using the function `graph.chi2()`. This function re-evaluates all residuals and calculates the current error. Plot in a graphic its result w.r.t time.

B. (10 pts) **Visualization.** Plot the current trajectory and landmark estimates in the `run.py` file.

Note: you may need to create a data structure to keep track of the landmarks id's to plot them separately and all the state variables corresponding to poses.

C. (5 pts) **Adjacency matrix.** Plot the current adjacency matrix at the last time step. For this, use the function `graph.get_adjacency_matrix()`, returning a sparse matrix. Comment on its structure. Also print the information matrix using the function `graph.get_information_matrix()`.

Note: use matplotlib `spi` function

D. (10 pts) **Covariance.** Plot the 3-sigma iso-contour of the covariance of the last pose.

Note: you may use the function `plot2cov` in tools.

E. (10 pts) **Batch solution.** Disable solving solution at each iteration and solve only in the last time step. In this case, you would need to call multiple times `graph.solve()`, checking for convergence with the chi2 function. Fortunately, the Levenberg-Marquard algorithm `graph.solve(mrob.LM)` optimizes until convergence and adapts the update value of state variables as the iterations go on. Report on the number of iterations required (printed in console) and the final chi2 error achieved.